

DTIC FILE COPY

(2)

AVF Control Number: AVF-VSP-169.0788
~~88-01-25-ACI~~

AD-A199 608

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880620W1.09070
Apollo Computer, Inc.
Domain/Ada, Version 2.0
Apollo DN4000

Completion of On-Site Testing:
22 June 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

DTIC
ELECTE
SEP 26 1988
S H D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

88 9 26 04

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report; Apollo Computer, Inc., Domain/Ada, Version 2.0, Apollo DN4000 (Host and Target).		5. TYPE OF REPORT & PERIOD COVERED 22 June 1988 to 22 June 1989
7. AUTHOR(s) Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.		12. REPORT DATE 22 June 1988
		13. NUMBER OF PAGES 54 p.
		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report) UNCLASSIFIED		
18. SUPPLEMENTARY NOTES		
19. KEYWORDS (Continue on reverse side if necessary and identify by block number) Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Domain/Ada, Version 2.0, Apollo Computer, Inc., Wright-Patterson Air Force Base, Apollo DN4000 under Domain/IX, Release SR9.7 (Host and Target), ACVC 1.9.		

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: Domain/Ada, Version 2.0

Certificate Number: 880620W1.09070

Host:

Apollo DN4000 under
Domain/IX,
Release SR9.7

Target:

Apollo DN4000 under
Domain/IX,
Release SR9.7

Testing Completed 22 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.



Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311



Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT	1-2
1.3	REFERENCES	1-3
1.4	DEFINITION OF TERMS	1-3
1.5	ACVC TEST CLASSES	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED	2-1
2.2	IMPLEMENTATION CHARACTERISTICS	2-2
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER	3-2
3.4	WITHDRAWN TESTS	3-2
3.5	INAPPLICABLE TESTS	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS	3-3
3.7	ADDITIONAL TESTING INFORMATION	3-4
3.7.1	Prevalidation	3-4
3.7.2	Test Method	3-4
3.7.3	Test Site	3-5
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 22 June 1988 at Apollo Computer, Inc.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

INTRODUCTION

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures and Guidelines</u> .

INTRODUCTION

AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is

INTRODUCTION

passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

INTRODUCTION

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Domain/Ada, Version 2.0

ACVC Version: 1.9

Certificate Number: 880620W1.09070

Host Computer:

Machine: Apollo DN4000

Operating System: Domain/IX
Release SR9.7

Memory Size: 8 megabytes

Target Computer:

Machine: Apollo DN4000

Operating System: Domain/IX
Release SR9.7

Memory Size: 8 megabytes

CONFIGURATION INFORMATION

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- . Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- . Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- . Predefined types.

This implementation supports the additional predefined types `SHORT_INTEGER`, `TINY_INTEGER`, and `SHORT_FLOAT` in the package `STANDARD`. (See tests B86001C and B86001D.)

- . Based literals.

An implementation is allowed to reject a based literal with a value exceeding `SYSTEM.MAX_INT` during compilation, or it may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` during execution. This implementation raises `NUMERIC_ERROR` during execution. (See test E24101A.)

- . Expression evaluation.

Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

CONFIGURATION INFORMATION

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently `NUMERIC_ERROR` is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

No exception is raised when a literal operand in a fixed-point comparison test is outside the range of the base type. (See test C45252A.)

Apparently `NUMERIC_ERROR` is raised when a literal operand in a fixed-point membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

. Rounding.

The method used for rounding to integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round to even. (See test C4A014A.)

. Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`. For this implementation:

Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises no exception. (See test C36003A.)

`NUMERIC_ERROR` is raised when `'LENGTH` is applied to an array type with `INTEGER'LAST + 2` components. (See test C36202A.)

`NUMERIC_ERROR` is raised when `'LENGTH` is applied to an array type with `SYSTEM.MAX_INT + 2` components. (See test C36202B.)

A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `NUMERIC_ERROR` when the array type is declared. (See test C52103X.)

CONFIGURATION INFORMATION

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array subtype is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

- Pragmas.

The pragma `INLINE` is supported for procedures. The pragma `INLINE` is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package `SEQUENTIAL_IO` can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package `DIRECT_IO` can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

CONFIGURATION INFORMATION

Modes `IN_FILE` and `OUT_FILE` are supported for `SEQUENTIAL_IO`. (See tests CE2102D and CE2102E.)

Modes `IN_FILE`, `OUT_FILE`, and `INOUT_FILE` are supported for `DIRECT_IO`. (See tests CE2102F, CE2102I, and CE2102J.)

`RESET` and `DELETE` are supported for `SEQUENTIAL_IO` and `DIRECT_IO`. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for `SEQUENTIAL_IO` and `DIRECT_IO`. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in `OUT_FILE` mode, can be created in `OUT_FILE` mode, and can be created in `IN_FILE` mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107F..I (5 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file can be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file can be deleted for `SEQUENTIAL_IO`, `DIRECT_IO`, and `TEXT_IO`. (See test CE2110B.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

. Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

CONFIGURATION INFORMATION

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 226 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 26 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	109	1049	1630	17	18	46	2869
Inapplicable	1	2	223	0	0	0	226
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

TEST INFORMATION

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14		
Passed	190	499	540	245	166	98	142	326	137	36	234	3	253	2869	
Inapplicable	14	73	134	3	0	0	1	1	0	0	0	0	0	226	
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27	
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122	

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C87B04B	C85018B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 226 tests were inapplicable for the reasons indicated:

- C35702B uses LONG_FLOAT which is not supported by this implementation.
- A39005G uses a record representation clause which allocates only four bits for a component that is declared as a boolean array with four members. This implementation requires that the boolean array type have a separate size clause specifying four bits.

TEST INFORMATION

- The following 13 tests use `LONG_INTEGER`, which is not supported by this implementation:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.
- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.
- C86001F redefines package `SYSTEM`, but `TEXT_IO` is made obsolete by this new definition in this implementation and the test cannot be executed since the package `REPORT` is dependent on the package `TEXT_IO`.
- C96005B requires the range of type `DURATION` to be different from those of its base type; in this implementation they are the same.
- The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

TEST INFORMATION

The following 26 Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A	B24204A	B24204B	B24204C	B2A003A
B2A003B	B2A003C	B33301A	B37201A	B38003A
B38003B	B38009A	B38009B	B41202A	B44001A
B64001A	B67001A	B67001B	B67001C	B67001D
B91001H	B91003B	B95001A	B97102A	BC1303F
BC3005B				

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the Domain/Ada compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Domain/Ada using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of six Apollo DN4000 machines operating under Domain/IX, Release SR9.7.

A magnetic tape containing all tests except for the withdrawn tests and the tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled and linked on the Apollo DN4000, and all executable tests were run on the DN4000. Results were printed from the host computer.

The compiler was tested using command scripts provided by Apollo Computer, Inc., and reviewed by the validation team. The compiler was tested using all default option settings.

Tests were compiled, linked, and executed (as appropriate) using a single host/target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

TEST INFORMATION

3.7.3 Test Site

Testing was conducted at Apollo Computer, Inc., and was completed on 22 June 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

Apollo Computer, Inc. has submitted the following
Declaration of Conformance concerning the Domain/Ada
compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Apollo Computer, Inc.
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version 1.9

Base Configuration


Base Compiler Name: Domain/Ada Version 2.0

Host Architecture ISA: Apollo DN4000
OS & VER #: Domain/IX, Release SR9.7

Target Architecture ISA: Apollo DN4000
OS & VER #: Domain/IX, Release SR 9.7

Implementor's Declaration

I, the undersigned, representing Apollo Computer, Inc., have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Apollo Computer, Inc. is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.



Date: 1/22/88

Apollo Computer, Inc.

Daryl R. Winters, Ada Project Engineer

Owner's Declaration

I, the undersigned, representing Apollo Computer, Inc., take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



Date: 1/22/88

Apollo Computer, Inc.

Daryl R. Winters, Ada Project Engineer

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Domain/Ada, V2.0, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

package STANDARD is

...

type INTEGER is range -2147483648 .. 2147483647;

type SHORT_INTEGER is range -32758 .. 32767;

type TINY_INTEGER is range -128 .. 127;

type FLOAT is digits 9

range -1.79769313486231E+308 .. 1.79769313486231E+308;

type SHORT_FLOAT is digits 6 range -3.40282E+38 .. 3.40282E+38;

type DURATION is delta 2.0 ** (-14) range -86400.0 .. 86400.0;

...

end STANDARD;

15.3 Pragmas and Their Effects

Each of this implementation's pragmas is briefly described here; additional information on some of them is found under discussions of particular language constructs.

`pragma CONTROLLED` is recognized by the implementation, but has no effect in the current release.

`pragma ELABORATE` is implemented as described in Appendix B of the RM.

`pragma EXTERNAL_NAME` allows the user to specify a *link_name* for an Ada variable or subprogram so that the object can be referenced from other languages. For more information, see Chapter 11.

`pragma IMPLICIT_CODE` specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF) and is used only within the declarative part of a machine code procedure. For more information, see Chapter 10.

`pragma INLINE` is implemented as described in Appendix B of the RM with the addition that recursive calls can be expanded with the pragma up to the maximum depth of 8. Warnings are produced for too-deep nestings or for bodies that are not available for inline expansion.

`pragma INLINE_ONLY` when used in the same way as `pragma INLINE`, indicates to the compiler that the subprogram must *always* be inlined (very important for some code procedures.). This pragma also suppresses the generation of a callable version of the routine which saves code space.

`pragma INTERFACE` supports calls to Domain C, Domain Pascal, and FORTRAN language functions with an optional linker name for the subprogram. The Ada specifications can be either functions or procedures. All parameters must have mode IN. For more information, see Chapter 11.

`pragma INTERFACE_OBJECT` allows variables defined in another language to be referenced directly in Ada, replacing all occurrences of *variable_name* with an external reference to *link_name* in the object file. For more information, see Chapter 11.

`pragma LIST` is implemented as described in Appendix B of the RM.

`pragma MEMORY_SIZE` is recognized by the implementation, but has no effect in the current release.

`pragma NO_IMAGE` suppresses the generation of the image array used for the `IMAGE` attribute of enumeration types. This eliminates the overhead required to store the array in the executable image.

`pragma OPTIMIZE` is recognized by the implementation, but has no effect in the current release. See the `ada -O` option for code optimization options.

`pragma PACK` will cause the compiler to minimize gaps between components in the representation of composite types. For arrays, components will only be packed to bit sizes corresponding to powers of 2 (if the field is smaller than `STORAGE_UNIT` bits). Objects larger than a single `STORAGE_UNIT` are packed to the nearest `STORAGE_UNIT`.

`pragma PAGE` is implemented as described in Appendix B of the RM. It is also recognized by the source code formatting tool `a.pr`.

`pragma PRIORITY` is implemented as described in Appendix B of the RM.

`pragma SHARE_CODE` provides for the sharing of object code between multiple instantiations of the same generic procedure or package body. A 'parent' instantiation is created and subsequent instantiations of the same types can share the parent's object code, reducing program size and compilation times. The name `pragma SHARE_BODY` may be used instead of `SHARE_CODE` with the same effect.

`pragma SHARED` is recognized by the implementation, but has no effect in the current release.

`pragma STOPAGE_UNIT` is recognized by the implementation, but has no effect in the current release. The implementation does not allow `SYSTEM` to be modified by means of pragmas. However, the same effect can be achieved by recompiling package `SYSTEM` with altered values.

`pragma SUPPRESS` is supported in the single parameter form. The pragma applies from the point of occurrence to the end of the innermost enclosing block. `DIVISION_CHECK` cannot be suppressed. The double parameter form of the pragma with a name of an object, type, or subtype is recognized, but has no effect in the current release.

`pragma SYSTEM_NAME` is recognized by the implementation, but has no effect in the current release. The implementation does not allow `SYSTEM` to be modified by means of pragmas. However, the file `system.a` from the `STANDARD` library can be copied to a local Domain/Ada library and recompiled there with new values.

15.4 Implementation-Defined Attribute: `X'REF`

Domain/Ada provides one implementation-defined attribute, `'REF`. There are two forms of use for this attribute, `X'REF` and `SYSTEM.ADDRESS'REF(N)`. `X'REF` is used only in machine code procedures while `SYSTEM.ADDRESS'REF(N)` can be used anywhere to convert an integer expression to an address.

15.4.1 `X'REF`

The attribute generates a reference to the entity to which it is applied.

In `X'REF`, `X` must be either a constant, variable, procedure, function, or label. The attribute returns a value of the type `MACHINE_CODE.OPERAND` and may only be used to designate an operand within a code-statement.

The instruction generated by the code-statement in which the attribute occurs may be preceded by additional instructions needed to facilitate the reference (for example, loading a base register). If the declarative section of the procedure contains `pragma IMPLICIT_CODE (OFF)`, a warning will be generated if additional code is required.

References may also cause run-time checks to be generated. `pragma SUPPRESS` may be used to eliminate these checks:

```
CODE_1'(JSR, PROC'REF);  
CODE_2'(MOVE_L, X.ALL(Z)'REF, DO);
```

For more information, see Chapter 10.

15.4.2 SYSTEM.ADDRESS'REF(N)

The effect of this attribute is similar to the effect of an unchecked conversion from integer to address. However, SYSTEM.ADDRESS'REF(N) should be used instead in the following listed circumstances, and in these circumstances, N must be static.

- Within any of the run time configuration packages.
Use of unchecked conversion within an address clause would require the generation of elaboration code, but the configuration packages are not elaborated.
- In any instance where N is greater than INTEGER'LAST.
Such values are required in address clauses which reference the upper portion of memory. To use unchecked conversion in these instances would require that the expression be given as a negative integer.
- To place an object at an address, use the 'REF attribute.
The *integer_value*, in the example below, is converted to an address for use in the address clause representation specification. The form avoids UNCHECKED_CONVERSION and is also useful for 32-bit unsigned addresses.

—place an object at an address
for object use at ADDRESS'REF (*integer_value*)

—to use unsigned addresses
for VECTOR use at SYSTEM.ADDRESS'REF(16#808000d0#);
TOP_OF_MEMORY: SYSTEM.ADDRESS:= SYSTEM.ADDRESS'REF(16#FFFFFFFF#);

In SYSTEM.ADDRESS'REF(N), SYSTEM.ADDRESS must be the type SYSTEM.ADDRESS. N must be an expression of type UNIVERSAL_INTEGER. The attribute returns a value of type SYSTEM.ADDRESS, which represents the address designated by N.

15.5 Restrictions on 'Main' Programs

Domain/Ada requires that a 'main' program must be a non-generic subprogram that is either a procedure or a function returning an Ada `STANDARD.INTEGER` (the predefined type). In addition, a 'main' program cannot be an instantiation of a generic subprogram.

15.6 Generic Declarations

Domain/Ada does not require that a generic declaration and the corresponding body be part of the same compilation, and they are not required to exist in the same Domain/Ada library. An error is generated if a single compilation contains two versions of the same unit.

15.7 Shared Object-Code for Generic Subprograms

The Domain/Ada compiler generates code for a generic instantiation that can be shared by other instantiations of the same generic. This reduces the size of the generated code and increasing compilation speed. There is an overhead associated with the use of shared code instantiations because the generic actual parameters must be accessed indirectly and in the case of a generic package instantiation, declarations in the package are also accessed indirectly. Also, greater optimization is possible for unshared instantiations because exact actual parameters are known. It is the responsibility of the programmer to decide whether space or time is most critical in a specific application.

To give the programmer control of when an instantiation generates unique code or shares code with other similar instantiations, we provide `pragma SHARE_CODE`. This pragma can be applied to a generic declaration or to individual instantiations.

It is not practical to share the code for instantiations of all generics. If the generic has a formal private type parameter the generated code to accommodate an instantiation with an arbitrary actual type would be extremely inefficient.

The Domain/Ada compiler will share code by default if the generic formal type parameters are restricted to integer, enumeration, or floating point. To override the default, the `pragma SHARE_CODE(name, FALSE)` must be used. If there are formal subprogram parameters, instantiations will not be shared unless an explicit `pragma SHARE_CODE(name, TRUE)` is used.

Generics are shared by default, if a parent is visible, except in the following cases:

- When generic formal types other than integer, enumeration, `SYSTEM.ADDRESS` or floating point are used
- When `pragma INLINE` is applied to a generic subprogram or instantiation or to a subprogram visible at the library level within a generic package or instantiation
- When the representations of the actual type parameters are not the same for each of the instantiations
- When the generic has a formal in out parameter and the subtype of the corresponding actual is not the same as the subtype of the formal parameter

The `pragma SHARE_CODE` is used to indicate desire to share or not share an instantiation. The pragma can reference either the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on or off for all instantiations of that generic unless overridden by specific `SHARE_CODE` pragmas for individual instantiations. When it references an instantiated unit, sharing is on or off only for that unit. The default is to share all generics that can be shared unless the unit uses `pragma INLINE`.

The `pragma SHARE_CODE` is only allowed in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is

```
pragma SHARE_CODE (generic_name, boolean_literal)
```

Note that a parent instantiation (the instantiation that creates the shareable body) is independent of any individual instantiation, therefore reinstantiation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

We've compiled a unit, `SHARED_IO`, in the standard library that instantiates all Ada generic I/O packages for the most commonly used base types. Thus, any instantiation of an Ada I/O generic package will share one of the parent instantiation generic bodies unless

```
pragma SHARE_CODE ( generic_name, FALSE );
```

is given.

15.8 Representation Specifications

Representation Clauses — Domain/Ada supports bit-level representation clauses.

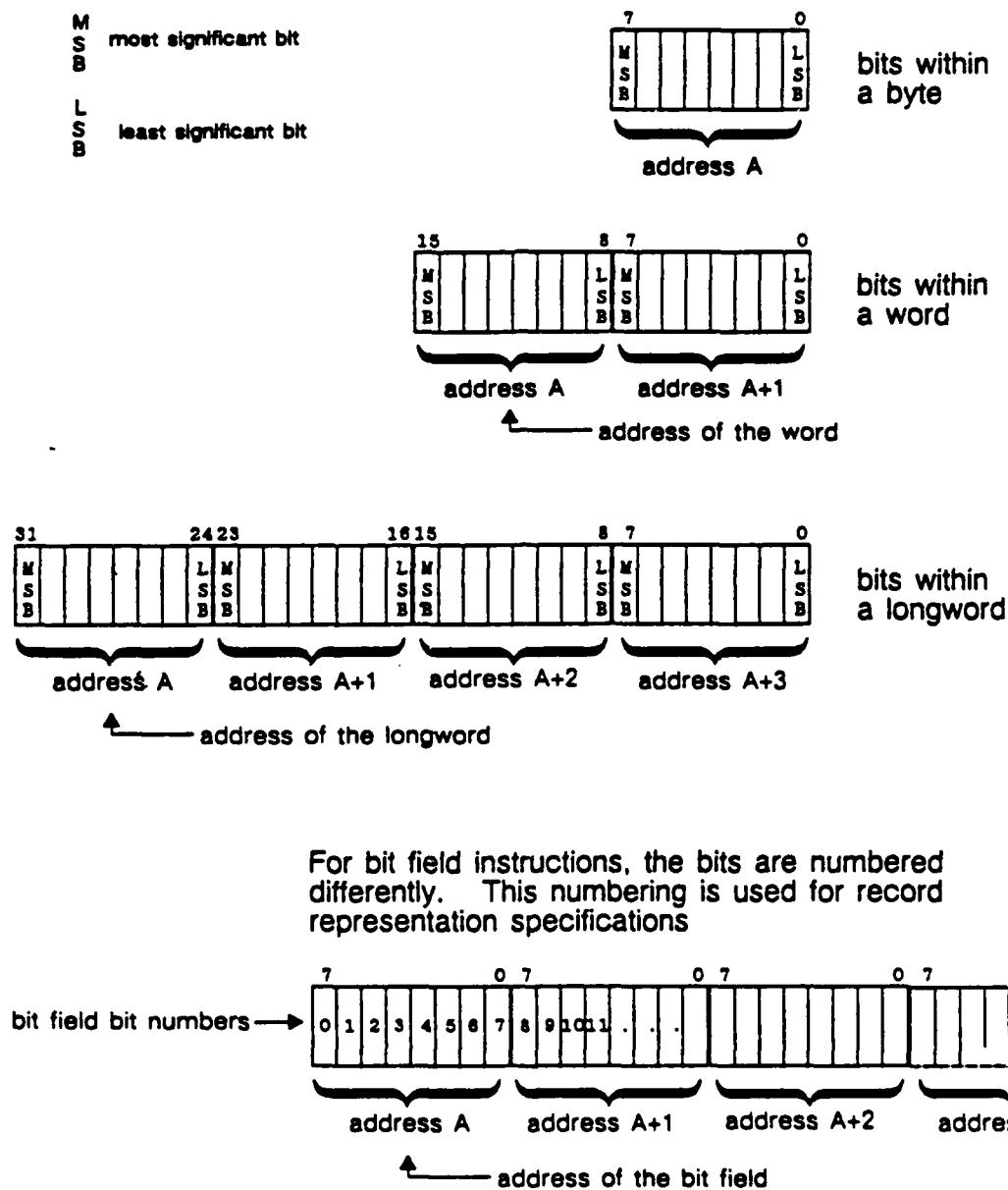
pragma PACK — Objects and components are packed to the nearest power of two bits. Domain/Ada does not define any additional representation pragmas.

Length Clauses — Domain/Ada supports all representation clauses.

Enumeration Representation Clauses — Enumeration representation clauses are supported.

Record Representation Clauses — Representation specifications are based on the target machine's word, byte, and bit order numbering so that Domain/Ada is consistent with various machine architecture manuals. Bits within a `STORAGE_UNIT` are also numbered according to the target machine manuals. It is not necessary for a user to understand the default layout for records and other aggregates since fine control over the layout is obtained by the use of record representation specifications. It is then possible to align record fields correctly with structures and other aggregates from other languages by specifying the location of each element explicitly. Note that bit fields are numbered opposite the ordering for bits within a byte on M68000 family processors. Bit fields use the numbering specified for the MC68020 bit extraction instructions. The `'FIRST_BIT` and `'LAST_BIT` attributes can be used to construct bit manipulation code that is applicable to differently bit-numbered systems. Refer to the M68000 Family addressing and bit numbering illustration in Figure 15-2.

M68000 Family Addressing and Bit Numbering*



* Motorola, "2.3 Data Organization in Memory," *MC68020 32-Bit Microprocessor User's Manual*, p. 2-2. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Figure 15-2. M68000 Family Addressing and Bit Numbering

The only restrictions on record representation specifications are the following: if a component does not start and end on a storage unit boundary, it must be possible to get the component into a register with one move instruction. On a MC68000 machine, where longwords start on even bytes, it must fit into 4 bytes starting on a word boundary. For example, the following specification is illegal:

```
for REC use record at mod 2;
  FIELD at 1 range 2 .. 25;
    — Extraction of FIELD must start at byte 0 on MC68000. Byte 1,
    — bit 25 is 34 bits beyond the start of byte 0 (34 bits > 4
    — bytes, so cannot extract).
end record;
```

Also, a component that is itself a record must occupy a power of 2 bits. Components that are of a discrete type or packed array can occupy an arbitrary number of bits, subject to the above restrictions.

Note that in the example above a size specification could be given,

```
for REC'size use 39;
```

but due to alignment, such a record would always take 5 bytes (that is, 40 bits).

Address Clauses — Address clauses are supported for objects and entries.

For more information, see Section 15.4.2.

Change of Representation — Change of representation is supported.

package SYSTEM — For the specification of package SYSTEM, Section 15.14.1. This specification is also available online in the file `system.a` in the release standard library. The pragmas `SYSTEM_NAME`, `STORAGE_UNIT`, and `MEMORY_SIZE` are recognized by the implementation, but have no effect. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling the SYSTEM package with altered values. Note that such a compilation will cause other units in the STANDARD library to become out of date. Consequently, such recompilations should be made in a library other than `standard`.

System-Dependent Named Numbers — For the specification of package SYSTEM, see Section 15.14.1. This specification is also available online in the file `system.a` in the release standard library.

Representation Attributes — The `'ADDRESS` attribute is supported for the following entities:

- Variables
- Constants
- Procedures
- Functions

All other representation attributes are supported.

Representation Attributes of Real Types — These attributes are supported. For more information, see Section 15.14.

Machine Code Insertions — Machine code insertions are supported. For more information, see Chapter 10.

- **Interface to Other Languages** — For detailed information, refer to Chapter 11.

Unchecked Programming — Both `UNCHECKED_DEALLOCATION` and `UNCHECKED_CONVERSION` are provided.

Unchecked Storage Deallocations — Any object that was allocated may be deallocated. No checks are currently performed on released objects.

Unchecked Type Conversions — The predefined generic function `UNCHECKED_CONVERSION` cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

15.9 Source File Structure/Restrictions

Character Set — Domain/Ada provides the full *graphic_character* textual representation for programs. The character set for source files and internal character representations is ASCII.

Lexical Elements, Separators, and Delimiters — Domain/Ada uses normal Domain/OS I/O text files as input. Each line is terminated by a newline character (ASCII.LF).

Source File Limits —

- 499 characters per source line
- 1296 Ada units per source file
- 32767 lines per source file

Compiler/Tool Limits —

- 499 characters in identifiers and literals
- 4,000,000 STORAGE_UNITS in a statically sized record type
- 10,240 STORAGE_SIZE default for a task
- 100,000 STORAGE_UNITS default collection size for access type
- no limit number of declared objects (except virtual space)
- 800 characters in a rooted name (full path of an object)
- 8 number of recursive inlines
- 8 number of nested inlines
- 400 number of nested constructs
- 2048 characters in ADAPATH (library search list)
- 2048 characters in a WITH or INFO directive
- 16M memory use per compilation (other Domain/OS limits may apply)
- 50 lexical errors before the front end exits
- 100 syntax errors before the front end exits
- 10 attempts to lock GVAS_table
- 10 attempts to lock ada.lib
- 20 attempts to lock gnrx.lib
- 64 debugger breakpoints
- 32 debugger array dimensions in a p command
- 9 debugger 'call parameters'
- 256 debugger 'run parameters'

15.10 Parameter Passing

Parameters are passed by pushing values (or addresses) on the stack. Extra information is passed for records ('CONSTRAINED') and for arrays (dope vector address).

Small results are returned by value in registers; large results are passed by reference.

The compiler assumes the following calling conventions.

1. Caller pushes arguments on stack in reverse order from their declaration
2. Caller calls callee
3. Callee builds display and allocates space for local variables via LINK instruction
4. Callee pushes any registers it uses in the sets D2-D7, A2-A6, and FP2-FP7
5. Callee executes
6. Callee pops registers pushed in step 4
7. Callee leaves result in D0, A0, or FP0 if callee is a function
8. Callee deallocates local variables (via the UNLK instruction)
9. Callee returns to caller using the RTS instruction.
10. Caller copies back any out parameters or function value
11. Caller deallocates space used for arguments on the stack

Caution: Compilers for other languages may follow calling conventions other than those expected by Domain/Ada. The Domain/Ada debugger should be used to verify that the call interface is as expected.

Machine code insertions can be used to explicitly build a call interface when compiler conventions are not compatible, or when interfacing to assembly language.

For example, suppose an interface to a C function `pass_flt` is desired, where the C compiler generated code such that the caller allocates space for the return value:

```
float pass_flt(x);  
  int x;  
  { ...  
  }
```

The following Ada code would provide a wrapper to call this function:

```
with MACHINE_CODE;
function PASS_FLT(X : INTEGER) return SHORT_FLOAT is
RETURN_VAL : SHORT_FLOAT;

procedure WRAPPER is
  use MACHINE_CODE;
begin
  CODE_1'(PEA_L, RETURN_VAL'REF);
  CODE_2'(MOVE_L, X'REF, decr(sp));    -- push x onto the stack
  CODE_1'(JSR, EXT("pass_flt"));      -- call pass_flt via its
                                      -- link name

  CODE_2'(ADDQ_L, IMMED(8), sp);      -- save result
end WRAPPER;

pragma INLINE(WRAPPER);
begin
  WRAPPER;
  return SHORT_FLOAT'(RETURN_VAL)
end PASS_FLT;
```

◆ For more information, see Chapter 10.

15.11 Conversion and Deallocation

The predefined generic function `UNCHECKED_CONVERSION` cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

There are no restrictions on the types with which generic function `UNCHECKED_DEALLOCATION` can be instantiated. No checks are performed on released objects.

15.12 Process Stack Size

The compiler and other large dynamic compiled programs can occasionally give problems due to the shell's stack limit. Altering the stack size and recompiling or re-executing is sometimes necessary. A process inherits its stack limit from the invoking process, usually the shell.

In Domain/OS, the upper limit of the stack is determined by the address space layout; thus, Domain/OS processes can obtain a maximum of 256K of the stack. To change the stack size from the C shell, execute the following command (SR10 only):

`limit stacksize number`

Bourne shell implementations do not permit the stack size to be altered.

15.13 Interface Programming

pragma INTERFACE — This pragma supports calls to Domain C, Domain Pascal, FORTRAN, and UNCHECKED with an optional link name for the subprogram. The Ada specifications can be either functions or procedures. All parameters must have mode *in*.

For Domain C, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM.ADDRESS. Record and array objects can be passed by reference using the 'ADDRESS attribute.

For Domain Pascal, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS in SYSTEM.ADDRESS. Record and array objects can be passed by reference using the 'ADDRESS attribute.

For FORTRAN, all parameters are passed by reference; the parameter types must have type SYSTEM.ADDRESS. The result type for a FORTRAN function *must* be a scalar type.

UNCHECKED may be used to interface to an unspecified language, such as assembly language. The compiler will generate the call as if it were to an Ada procedure, but will not expect a matching Ada procedure body.

The optional *link* name enables calling a function whose name is defined in another language, allowing characters in the name that are not allowed in an Ada identifier. Case sensitivity can then be preserved. Without the optional link name, the Ada compiler converts all identifiers to lower case. The link name overrides the default transformations that **pragma INTERFACE** performs on the name to create the unresolved reference name in the object module.

`pragma INTERFACE_OBJECT` allows variables defined in another language to be referenced directly in Ada. `pragma INTERFACE_OBJECT` replaces all occurrences of *variable_name* with an external reference to *linker_name* in the object file using the format shown below:

```
pragma INTERFACE_OBJECT (variable_name, "linker_name");
```

This pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object *cannot* be any of the following:

- Loop variable
- Constant
- Initialized variable
- Array
- Record

The *linker_name* must be constructed as expected by the linker. The example below makes the C global variable `errno` available within an Ada program:

```
package PACKAGE_NAME is
    ...
    ERRNO: integer;
    pragma INTERFACE_OBJECT (ERRNO, "errno");
    ...
end PACKAGE_NAME;
```

`pragma EXTERNAL_NAME` allows the user to specify a *linker_name* for an Ada variable or subprogram so that the Ada object can be referenced from other languages using the syntax shown below:

```
pragma EXTERNAL_NAME (object_or_subprogram_name, "linker_name");
```

Objects must be variables defined in a package specification; subprograms can be either library level or within a package specification.

For more information, see Chapter 11.

15.14 Predefined Packages and Generics

The following predefined Ada packages given by RM Appendix C(22) are provided in the standard library:

- package STANDARD
- package CALENDAR
- package SYSTEM
- generic procedure UNCHECKED_DEALLOCATION
- generic function UNCHECKED_CONVERSION
- generic package SEQUENTIAL_IO
- generic package DIRECT_IO
- package TEXT_IO
- package IO_EXCEPTIONS
- package LOW_LEVEL_IO
- package MACHINE_CODE

The implementation dependent portions of the packages define the following types and objects:

— in package STANDARD

```
type BOOLEAN is          <8-bit, byte>;
type TINY_INTEGER is     <8-bit, byte integer>;
type SHORT_INTEGER is    <16-bit, word integer>;
type INTEGER is          <32-bit, longword integer>;
type SHORT_FLOAT is      <6-digit, 32-bit, float>;
type FLOAT is            <9-digit, 64-bit, float>;
type DURATION is        delta 2.0**(-14) range -86400.0 .. +86400.0;
```

— in package DIRECT_IO

```
type COUNT is            range 0 .. 2_147_483_647;
```

— in package TEXT_IO

```
type COUNT is            range 0 .. 2_147_483_647;
subtype FIELD is         INTEGER range 0 .. INTEGER'last;
```

15.14.1 Specification of Package SYSTEM

```

package SYSTEM is
  type NAME is ( apollo_4_3_unix );
  SYSTEM_NAME      : constant NAME := apollo_4_3_unix;
  STORAGE_UNIT     : constant := 8;
  MEMORY_SIZE      : constant := 16_777_216;
  -- System-Dependent Named Numbers
  MIN_INT          : constant := -2_147_483_648;
  MAX_INT          : constant := 2_147_483_647;
  MAX_DIGITS       : constant := 15;
  MAX_MANTISSA     : constant := 31;
  FINE_DELTA       : constant := 2.0**(-31);
  TICK             : constant := 0.01;
  -- Other System-dependent Declarations
  subtype PRIORITY is INTEGER range 0 .. 99;
  MAX_REC_SIZE     : integer := 64*1024;
  type ADDRESS is private;
  NO_ADDR : constant ADDRESS;

  function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
  function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
  function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
  function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
  function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
  function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
  function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
  function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

  function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
  function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
  function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
  function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
  function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
  function "+"(A: ADDRESS;
               INCR: INTEGER) return ADDRESS renames INCR_ADDR;
  function "-"(A: ADDRESS;
               DECR: INTEGER) return ADDRESS renames DECR_ADDR;

  pragma inline(ADDR_GT);
  pragma inline(ADDR_LT);
  pragma inline(ADDR_GE);
  pragma inline(ADDR_LE);
  pragma inline(ADDR_DIFF);
  pragma inline(INCR_ADDR);
  pragma inline(DECR_ADDR);
  pragma inline(PHYSICAL_ADDRESS);

private
  type ADDRESS is new integer;
  NO_ADDR : constant ADDRESS := 0;
end SYSTEM;

```

15.14.2 Package CALENDAR

CALENDAR's clock function (in package CALENDAR.LOCAL_TIME located in the file `calendar_s.a`) uses the Domain/OS service routines GETTIMEOFDAY and LOCALTIME for getting the current time.

15.14.3 Package SEQUENTIAL_IO

Sequential I/O is currently implemented for variant records, but with the restriction that the maximum size possible for the record will always be written. This is also true of direct I/O. For unconstrained records and arrays, the constant, SYSTEM.MAX_REC_SIZE, can be set prior to the elaboration of the generic instantiation of SEQUENTIAL_IO or DIRECT_IO. For example, if unconstrained strings are written, SYSTEM.MAX_REC_SIZE effectively restricts the maximum size of string that can be written. If the user knows the maximum size of such strings, the SYSTEM.MAX_REC_SIZE may be set prior to instantiating SEQUENTIAL_IO for the string type. This variable can be reset after the instantiation with no effect.

15.15 Types, Ranges, and Attributes

Numeric Literals — Domain/Ada uses unlimited precision arithmetic for computations with numeric literals.

Enumeration Types — Domain/Ada allows an unlimited number of literals within an enumeration type.

Attributes of Discrete Types — Domain/Ada defines the image of a character that is not a graphic character as the corresponding 2-character or 3-character identifier from package ASCII of RM Annex C-4. The identifier is in uppercase without enclosing apostrophes. For example, the image for a carriage return is the 2-character sequence CR (ASCII.CR).

The type STRING — Except for memory size, Domain/Ada places no specific limit on the length of the predefined type STRING. Any type derived from the type STRING is similarly unlimited.

Integer Types — Domain/Ada provides three integer types in addition to *universal_integer*: INTEGER, SHORT_INTEGER, and TINY_INTEGER. Table 15-4 lists the ranges for these integer types.

Table 15-4. Domain/Ada Integer Types

Name of Attribute	Attribute Value of INTEGER	Attribute Value of SHORT_INTEGER	Attribute Value of TINY_INTEGER
FIRST	-2_147_483_648	-32_768	-128
LAST	2_147_483_647	32_767	127

Operation of Floating Point Types — Domain/Ada floating point types have the attributes listed in Table 15-5.

Table 15-5. Domain/Ada Floating-Point Types

Name of Attribute	Attribute Value of FLOAT	Attribute Value of SHORT_FLOAT
SIZE	64	32
FIRST	-1.79769313486231E+308	-3.40282E+38
LAST	1.79769313486231E+308	3.40282E+38
DIGITS	15	6
MANTISSA	51	21
EPSILON	8.88178419700125E-16	9.53674316406250E-07
EMAX	204	84
SMALL	1.94469227433160E-62	2.58493941422821E-26
LARGE	2.57110087081438E+61	1.93428038904620E+25
SAFE_EMAX	1022	126
SAFE_SMALL	1.11253692925360E-308	5.87747175411143E-39
SAFE_LARGE	4.49423283715578E+307	8.5075511654154E+37
MACHINE_RADIX	2	2
MACHINE_MANTISSA	53	24
MACHINE_EMAX	1024	128
MACHINE_EMIN	-1022	-126
MACHINE_ROUNDS	TRUE	TRUE
MACHINE_OVERFLOWS	TRUE	TRUE

Fixed Point Types — Domain/Ada provides fixed point types mapped to the supported integer sizes.

Operations of Fixed Point Types — Domain/Ada fixed point type DURATION has the attributes listed in Table 15-6.

Table 15-6. Attributes for the Fixed-Point Type DURATION

Name of Attribute	Attribute Value for DURATION
SIZE	32
FIRST	-2147483.648
LAST	2147483.647
DELTA	1.000000000000000E-03
MANTISSA	32
SMALL	9.765625000000000E-04
LARGE	4.19430399902343E_06
FORE	8
AFT	3
SAFE_SMALL	9.765625000000000E-04
SAFE_LARGE	4.19430399902343E+06
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOW	TRUE

15.16 Input/Output

The Ada I/O system is implemented using Domain/OS I/O. Both formatted and binary I/O are available. There are no restrictions on the types with which DIRECT_IO and SEQUENTIAL_IO can be instantiated, except that the element size must be less than a maximum given by the variable SYSTEM.MAX_REC_SIZE. This variable can be set to any value prior to the generic instantiation; thus, the user can use any element size. DIRECT_IO can be instantiated with unconstrained types, but each element will be padded out to the maximum possible for that type or to SYSTEM.MAX_REC_SIZE, whichever is smaller. No checking — other than normal static Ada type checking — is done to ensure that values from files are read into correctly sized and typed objects.

Domain/Ada file and terminal input-output are identical in most respects and differ only in the frequency of buffer flushing. Output is buffered (buffer size is 1024 bytes). The buffer is always flushed after each write request if the destination is a terminal.

The procedure `FILE_SUPPORT.ALWAYS_FLUSH` (*file_ptr*) will cause flushing of the buffer associated with *file_ptr* after all subsequent output requests. Refer to the source code for `file_spprt_b.a` in the standard library.

Instantiations of `DIRECT_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example, for unconstrained arrays such as a string where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_REC_SIZE` is defined in `SYSTEM` and can be changed before instantiating `DIRECT_IO` to provide an upper limit on the record size. The maximum size supported is $1024 * 1024 * \text{STORAGE_UNIT}$ bits. `DIRECT_IO` will raise `USE_ERROR` if `MAX_REC_SIZE` exceeds this absolute limit.

Instantiations of `SEQUENTIAL_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example, for unconstrained arrays such as `STRING` where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_REC_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `INTEGER_IO` to provide an upper limit on the record size. `SEQUENTIAL_IO` imposes no limit on `MAX_REC_SIZE`.

15.17 Machine Code Insertions

The general definition of package `MACHINE_CODE` provides an assembly language interface for the target machine including the necessary record types needed in the code statement, an enumeration type containing all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions. Also supplied (for use only in units that `WITH MACHINE_CODE`) are pragma `IMPLICIT_CODE` and the attribute `X'REF`.

Machine code statements take operands of type `OPERAND`, a private type that forms the basis of all machine code address formats for the target.

The general syntax of a machine code statement is

```
CODE_n'(opcode, operand [, operana]));
```

where *n* indicates the number of operands in the aggregate.

When there is a variable number of operands, they are listed within a subaggregate using the syntax shown below:

```
CODE_n'(opcode, (operand [, operand]));
```

In the example shown below, `code_2` is a record 'format' whose first argument is an enumeration value of type `OPCODE` followed by two operands of type `OPERAND`:

```
CODE_2'(move_1, a'ref, b'ref);
```

For those opcodes requiring no operands, named notation must be used.

For more information, see the Ada RM 4.3(4):

```
CODE_0'(op => opcode);
```

The *opcode* must be an enumeration literal (that is, it cannot be an object, attribute, or a rename). An *operand* can only be an entity defined in `MACHINE_CODE` or the `X'REF` attribute.

Arguments to any of the functions defined in `MACHINE_CODE` must be static expressions, string literals, or the functions defined in `MACHINE_CODE`.

X'REF — The `X'REF` attribute denotes the effective address of the first of the storage units allocated to the object. `X'REF` is not supported for a package, task unit, or entry. For details, see Section 15.4.

pragma IMPLICIT_CODE — The `IMPLICIT_CODE` pragma specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF) and is used only within the declarative part of a machine code procedure. Implicit code includes preamble and postamble code (such as, code used to move parameters from and to the stack). Use of `pragma IMPLICIT_CODE` does not eliminate code generated for run time checks, nor does it eliminate call/return instructions (these can be eliminated by `pragma SUPPRESS` and `pragma INLINE`, respectively). A warning is issued if OFF is used and any implicit code needs to be generated. This pragma should be used with caution.

As an example of machine code insertions, the procedure SET_ENABLE_CACHE_BIT is defined below. It sets F bit of the MC68020 CACR register:

```

-- freeze the cache
procedure set_enable_cache_bit is
  use machine_code;
begin
  code_2'(movec, cacr, d0);
  code_2'(or_1, +2#10#, d0);
  code_2'(movec, d0, cacr);
end;
pragma inline(set_enable_cache_bit);

procedure enable_cache is
  before_call, after_call: integer;
begin
  before_call := 1;
  set_enable_cache_bit;
  after_call := 1;
end;

```

Note that the machine code procedure is inline. The output of a.das excerpted below shows a procedure that calls SET_ENABLE_CACHE_BIT and the code generated for the call:

```

17      before_call := 1;    -- generate instruction before call
      03a: move.l      #01, (-0c,a6)
      8      code_2'(movec, cacr, d0);
      042: movec.l      cacr, d0
      9      code_2'(or_1, +2#10#, d0);
      046: or.l        #02, d0
     10      code_2'(movec, d0, cacr);
      04c: movec.l      d0, cacr
     19      after_call := 1;    -- generate instruction after call
      050: move.l      #01, (-010,a6)

```

For more information, see Chapter 10.

APPENDIX C
TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..498 => 'A', 499 => '1')
\$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..498 => 'A', 499 => '2')
\$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..249 => 'A', 250 => '3', 251..499 => 'A')
\$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..249 => 'A', 250 => '4', 251..499 => 'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..496 => '0', 497..499 => "298")

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..493 => '0', 494..499 => "69.0E1")
\$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1 => '"', 2..200 => 'A', 201 => '"')
\$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 => '"', 2..300 => 'A', 301..302 => "1'")
\$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..479 => ' ')
\$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
\$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
\$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	"/illegal/file_name/2{[]\$%2102C.DAT"
\$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	"/illegal/file_name/CE2102C*.DAT"
\$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10_000_000.0
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	"/no/such/directory/ILLEGAL_EXTERNAL_FILE_NAME1"
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/no/such/directory/ILLEGAL_EXTERNAL_FILE_NAME2"
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
\$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
\$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10_000_000.0
\$MAX_DIGITS Maximum digits supported for floating-point types.	15
\$MAX_IN_LEN Maximum input line length permitted by the implementation.	499
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
\$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..496 => '0', 497..499 => "11:")
\$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:", 4..495 => '0', 496..499 => "F.E:")
\$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..498 => 'A', 499 => '"')
\$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	TINY_INTEGER
\$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFFD#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- . B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- . E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);"; The Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- . C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.
- . C35502P: The equality operators in lines 62 and 69 should be inequality operators.
- . A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.
- . C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.
- . C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

WITHDRAWN TESTS

- . C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.
- . C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.
- . C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.
- . C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.
- . C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.
- . C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.
- . C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOW is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOW may still be TRUE.
- . C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.
- . A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.
- . BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.
- . AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.
- . CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.
- . CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.